Elaborato di Sistemi per il Governo dei Robot:

Robot di servizio in ambiente ospedaliero, analisi e confronto di algoritmi di motion planning

Andrea Cavaliere M58/240
Marco Caianiello M58/242
Pierluigi Porreca M58/259

Contents

List of Figures

Introduzione

Gli algoritmi di pianificazione del movimento sono utilizzati in molti campi, tra cui la bioinformatica, animazione dei personaggi, videogiochi AI, progettazione assistita da computer e produzione assistita da computer, progettazione architettonica, automazione industriale, robotica chirurgia, e navigazione con robot singolo e multiplo sia in due che in tre dimensioni. Il problema della pianificazione del movimento può essere identificato come: Data una posizione di partenza di il robot, una posa desiderata, una descrizione geometrica del robot e un descrizione geometrica del mondo, l'obiettivo è quello di trovare un percorso che muova il robot gradualmente dall'inizio alla fine, senza mai toccare alcun ostacolo. Il problema della pianificazione del movimento è PSPACE-hard, il che significa che qualsiasi algoritmo completo è computazionalmente difficile da analizzare. Al fine di affrontare questo problema e trovare una soluzione che permetta di ottenere un'analisi computazionale efficiente, ci sono diversi pratici algoritmi di pianificazione del movimento che rilassano i requisiti di completezza. La maggior parte degli algoritmi basati sul campionamento sono stati dimostratosi probabilisticamente completi, cioè la probabilità che l'algoritmo trovi una soluzione, se esiste, converge a uno come numero di campioni si avvicina all'infinito. Sono in grado anche nello spazio ad alte dimensioni di trovare un piano di movimento fattibile in tempi relativamente brevi, se si tratta di un'uscita. Nella pratica, gli algoritmi di pianificazione del movimento, includono la Probabilistic RoadMap (PRM), una delle categorie più significative di approcci alla pianificazione del movimento studiato in letteratura, e RRT che è un algoritmo pratico per la pianificazione del movimento sullo stato dell'arte delle piattaforme robotiche, perché può gestire sistemi con vincoli differenziali. L'algoritmo RRT ha la capacità di trovare una soluzione fattibile se esce, ma come hanno dimostrato Karaman e Frazzoli, se il numero di campioni aumenta la probabilità che l'algoritmo RRT converga verso una soluzione ottimale è in realtà zero. Questa caratteristica rende questo algoritmo meno utile nella pratica situazioni. Il problema potrebbe essere risolto con un metodo alternativo, RRT. un algoritmo basato sul campionamento con la pro-

prietà dell'ottimalità asintotica, cioè con quasi sicura convergenza verso una soluzione ottimale, insieme alla completezza probabilistica garanzie. RRT offre vantaggi sostanziali, soprattutto per le applicazioni in tempo reale. Come la RRT, la RRT trova rapidamente un piano di movimento fattibile.
Inoltre, migliora il piano verso la soluzione ottimale nel tempo rimanente prima che l'esecuzione del piano sia completata. Questa proprietà di perfezionamento è vantaggiosa, in quanto la maggior parte dei sistemi robotizzati richiede molto più tempo per l'esecuzione che pianificare le traiettorie. In tali impostazioni, l'ottimizzazione asintotica è particolarmente utile, dato che il tempo di calcolo disponibile come robot è mov3 lungo la sua traiettoria può essere utilizzato per migliorare la qualità delle rimanenti parte del percorso pianificato.
Un algoritmo di pianificazione del movimento in qualsiasi momento può esserein diversi modi. Il primo, proposto da Frazzoli in e Karaman adottato nel nostro progetto, si basa su criteri di ottimizzazione. Un sistema basato su questa pianificazione in qualsiasi momento si sovrappone a due funzioni nel tempo: l'esecuzione del suo piano attuale (qualche porzione iniziale di), e il calcolo da sostituire (qualsiasi pendente parte di) il piano attuale con un piano migliorato. Quindi dall'inizio il progettista, dopo il primo budget, trova una soluzione completa, in realtànon un percorso ottimale, che potrebbe portare il robot alla meta e durante il movimento lo migliora.
Al contrario l'altro metodo, proposto da Ferrari, Oriolo e Cognetti in[1] per qualsiasi intervallo di pianificazione, dà luogo solo a un soluzione parziale del movimento. L'algoritmo lascia intervalli di pianificazione ed esecuzione diversi: un movimento del corpo intero precedentemente pianificato viene eseguito mentre pianificare contemporaneamente una nuova soluzione per il successivo intervallo di esecuzione. Ad ogni intervallo di pianificazione, in modo diverso dal precedente paradigma,questo algoritmo si concentra sull'ottenimento della soluzione migliore, garantita per essere valida
nell'intervallo di esecuzione successivo, tra quelli che il pianificatore ha potuto per produrre entro il tempo di deliberazione, senza affidarsi a continui affinamenti della soluzione iniziale. I risultati complessivi devono essere costituiti da un sequenza di soluzioni a breve orizzonte calcolate on-line.

Chapter 1

Robotic Motion Planning

1.1 RRT

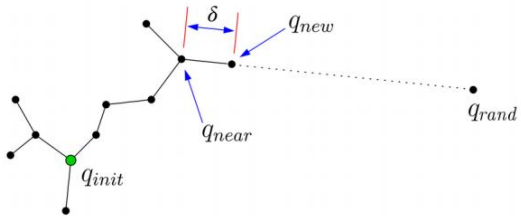


Figure 1.1: Processo di costruzione dell'albero di RRT

Rapdly-exploring Random Tree (RRT) è una struttura dati e un algoritmo di path planning che è progettato per la ricerca efficiente di percorsi in spazi non convessi ad alta dimensione. Le RRT sono costruite incrementalmente espandendo l'albero fino a un punto campionato a caso nello spazio di configurazione soddisfacendo determinati vincoli, ad esempio, incorporando ostacoli o dinamica vincoli. Mentre un algoritmo RRT può efficacemente trovare un percorso fattibile, un RRT da solo potrebbe non essere appropriato per risolvere un problema di pianificazione del percorso per un robot mobile in quanto non può incorporare informazioni aggiuntive sui costi, come la scorrevolezza o la lunghezza di il percorso. Pertanto, può essere considerato come un componente che può essere incorporato nel sviluppo di una varietà di diversi algoritmi di pianificazione, ad esempio RRT*.

```
GENERATE_RRT(x_init, K, Δt)
1  T.init(x_init);
2  for k = 1 to K do
3      x_rand ← RANDOM_STATE();
4      x_near ← NEAREST_NEIGHBOR(x_rand, T);
5      u ← SELECT_INPUT(x_rand, x_near);
6      x_new ← NEW_STATE(x_near, u, Δt);
7      T.add_vertex(x_new);
8      T.add_edge(x_near, x_new, u);
9  Return T
```

Figure 1.2: Pseudo code di un RRT

1.1.1 Algoritmo

x_i indica la posizione iniziale del robot in coordinate cartesiane. K indica il numero di vertici dell'albero e l'algoritmo sarà iterato K volte prima di terminare. La condizione di terminazione di questo loop può essere sostituita andando a verificare la distanza più vicina dal goal point all'albero. Δt indica l'intervallo di tempo e Υ rappresenta la struttura ad albero che contiene i campioni dei nodi dal configuration space e u indica l'ingresso di controllo.

- **Step 1** Si inizializza l'albero per avere il root alla posizione iniziale del robot
- **Step 2** Si sceglie una posizione random x_{rand} nel Configuration space.
- **Step 3** Si seleziona il vertice, x_{near} , nell'albero che è il più vicino a x_{rand}
- **Step 4** Viene selezionata una nuova configurazione, x_{new}, che si trova ad una distanza incrementale da x_{near} , nella direzione di x_{rand} .
- **Step 5** Si seleziona l'ingresso di controllo in grado di muovere i robot da x_{near} a x_{new}
- **Step 6** Si espande l'albero

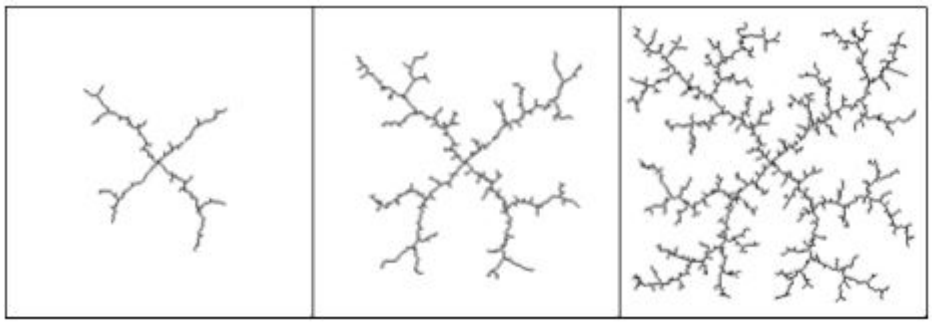


Figure 1.3: Esempio di costruzione di un RRT in un configuration space quadrato

L'RRT si espande rapidamente in poche direzioni per esplorare rapidamente i quattro angoli della piazza.
Anche se il metodo di costruzione è semplice, non è facile trovare un metodo che renda tale comportamento desiderabile. È perché un'espansione di una RRT è stata influenzata verso luoghi non ancora visitato

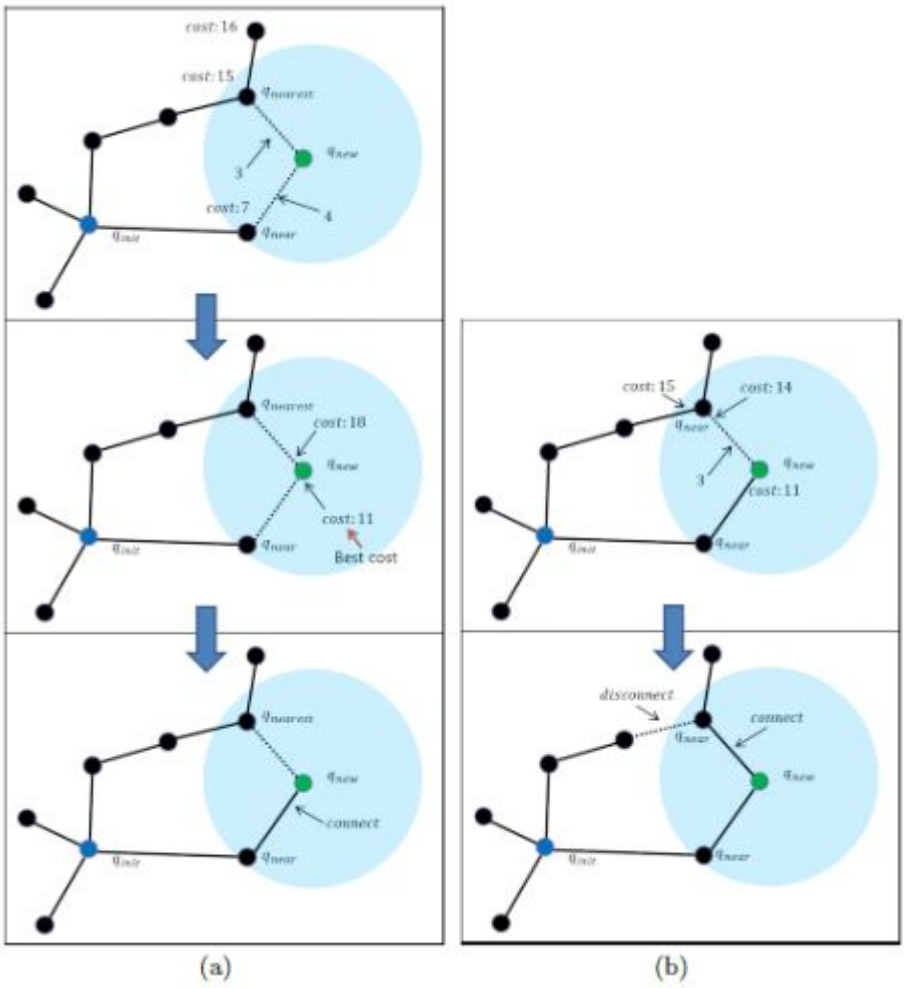


Figure 1.4: RRT* (a) ricerca del parent point (b) rewiring

1.2 RRT*

RRT* è algoritmo introdotto da Sertac Karaman e Emilio Frazzoli. Risolve il problema dell'algoritmo RRT che non converge a una soluzione ottimale di percorso con l'aumentare del numero di campioni. IL costo del cammino di RRT* converge a un valore ottimale, mantenendo una strutura ad albero come RRT.
RRT* eredita tutte le proprietà di RRT e lavora in maniera simile.
Aggiunge ad RRT due nuovi features chiamati "choose parent" e "rewire". *chooseparent* trova il miglior nodo parente per il nuovo nodo prima che questo sia inserito nell'albero. Questa ricerca è effettuata all'interno dell'area di una sfera di raggio k centrata in x_{new} con:

$$k = \gamma \left(\frac{log(n)}{n} \right)^{\frac{1}{d}}$$

dove d è lo distanza di ricerca e γ è la costante di planning basata sull'ambiente. $rewire$, invece, ricabla le connessioni dell'albero attraverso il raggio di area k per mantenere le connessioni dell'albero con il costo minimo. Con l'aumentare delle interazioni, RRT* migliora gradualmente il costo del percorso grazie alla proprietà di ottimizzazione asintotica. Il comportamento delle due funzioni è mostrato in Fig.1.4.

1.2.1 Choose parent & Rewire

Verranno presentate nel dettaglio le due nuove funzioni aggiuntive di RRT*:

- Invece di selezionare il nodo più vicino come genitore, RRT* guarda su tutti i nodi x_{near} in un intorno di x_{new} e ognuno di questi appartiene al sottospazio C_{near} . Infatti l'algoritmo calcola il costo nel selezionare tutti i possibili x_{near} . Questa operazione valuta il costo totale come la combinazione additiva del costo associato con il raggiungere il nodo parente e il costo della traiettoria x_{new} . Il nodo minimo x_{min} a cui è associato il costo più piccolo, diventa il nuovo "parent" ed è inserito nell'albero.

- L'algoritmo di rewire controlla ogni nodo x_{near} in prossimità di x_{new} per verificare se raggiungerlo tramite x_{new} avrebbe un costo inferiore rispetto quello che si farebbe con il current path. Quando questo rapporto diminuisce il costo relativo a x_{near}, l'algoritmo ricabla l'albero per rendere x_{new} il genitore di x_{near}

Grazie a queste due funzioni la qualità del percorso viene mogliorata senza incorrere in un sostanziale overhead computazionale e l'algoritmo converge ad una soluzione ottima quasi sicuramente. L'algoritmo è descritto nella figura che segue.

Algorithm 1: $T=(V,E) \leftarrow$ **RRT*** (q_{init})

```
1  T ← InitializeTree()
2  T ← InsertNode(∅, q_init, T)
3  for i=0 to i=N do
4       q_rand ← Sample(i)
5       q_nearest ← Nearest(T, q_rand)
6       (q_new, u_new, T_new) ← Steer(q_nearest, q_rand)
7       if ObstacleFree(q_new) then
8            Q_near ← Near(T, q_new, |V|)
9            q_min ← ChooseParent(Q_near, q_nearest, q_new, x_new)
10           T ← InsertNode(q_min, q_new, T)
11           T ← ReWire(T, Q_near, q_min, q_new)

12 Return T
```

Algorithm 2: $q_{min} \leftarrow$ **ChooseParent** $(C_{near}, q_{nearest}, x_{new})$

```
1  q_min ← q_nearest
2  c_min ← Cost(q_nearest)+c(x_new)
3  for q_near ∈ Cnear do
4       (x',u',T') ← Steer(q_nearest, q_rand)
5       if ObstacleFree(x') and x'(T')=q_new then
6            c'= Cost(q_near)+c(x')
7            if c'< Cost(q_new) and c'< c_min then
8                 q_min ← q_near;
9                 c_min ← c';

10 Return q_min
```

Algorithm 3: $T \leftarrow$ **ReWire** $(T, C_{near}, q_{min}, q_{new})$

```
1  for q_near ∈ Cnear:/{q_min} do
2       (x',u',T') ← Steer(q_nearest, q_rand)
3       if ObstacleFree(x') and x'(T')=q_near and
        Cost(q_new)+c(x') < Cost(q_near) then
4            T ← ReConnect(q_new, q_near, T)
5  Return T
```

Figure 1.5: Pseudo code di RRT*

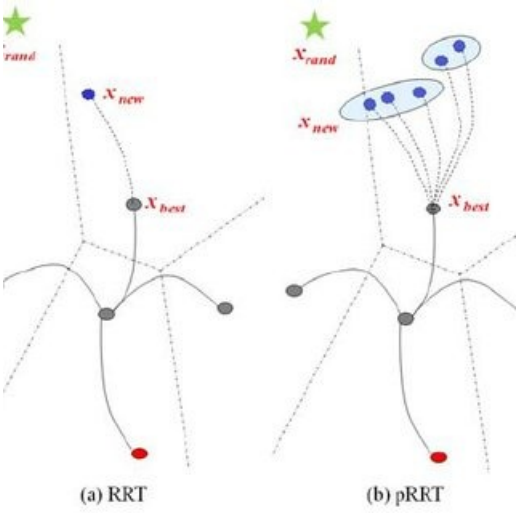


Figure 1.6: Parallel RRT

1.3 pRRT

Un metodo per migliorare le performance dell'algoritmo RRT è quello di sviluppare un'implementazione parallela, ottenendo il Parallel RRT. Attraverso un paradigma di "process splitting" si vanno ad applicare gli stessi problemi computazionali a diversi insieme, applicando lo stesso algoritmo randomizzato. Una volta che tutti quanti avranno terminato, si prenderà il migliore tra le soluzioni ottenute. Il Lock-Free Parallel RRT, ha un algoritmo quasi identico a quello dello standard RRT, ad eccezione del fatto che ogni thread campiona solo in una partizione dello spazio C e usa "lock-free nearest-neighborhood" data structure. In aggiunta prima di aggiungere il nuovo nodo nel path di RRT ai nodi vicini (quindi diventando visibile agli altri thread), processa un'adeguata operazione di recinzione di memoria per prevenire che altri thread possano osservare una vista parzialmente inizializzata del nuovo nodo aggiunto al path.

Chapter 2

Sviluppo e Simulazione

2.1 Scenario di Simulazione

Il progetto simula un ambiente ospedaliero in un cui un robot ha il compito di assistere medici e staff sanitario in operazioni di smistamento e consegna di materiale medico-sanitario, basti pensare alle operazioni di raccolta in magazzino di scorte e strumentazione oppure la consegna ai pazienti e medici di medicine e supporto logistico di qualsiasi genere.

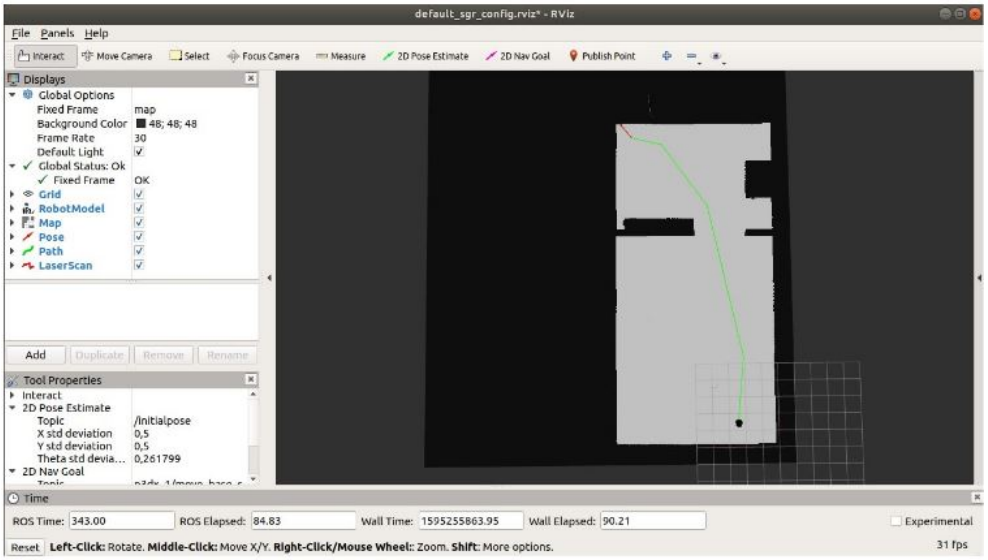
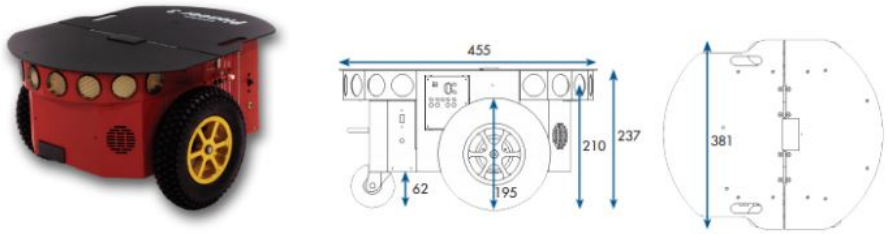


Figure 2.1: Esempio di traiettoria

A tale scopo il robot dovrà muoversi in ambiente particolarmente estesi e, per questione di semplicità, ipotizziamo uno scenario tipico quale un pi-



Parametri	Datasheet	In uso
Max. Forward/Backward Speed	1.2 m/s	0.3 m/s
Max Rotation Speed	5.24 rad/s	1.0 rad/s

Figure 2.2: Dati e misure del Pioneer 3-DX

ano di un ospedale di $1600m^2$ popolato da ostacoli mobili che tipicamente caratterizzano un ambiente del genere.

2.1.1 Pioneer 3-DX

Pioneer 3-DX è un piccolo robot leggero a due ruote motrici a due motori a trazione differenziale ideale per ambienti interni. Il modello è dotato di un sensore LiDAR 2D per interni, posizionato sulla parte anteriore della piastra, lasciando lo spazio necessario per trasportare oggetti di diverse dimensioni. Inoltre rispetto i valori di velocità lineare a rotazionale riportati sul datasheet sono state apportate delle modifiche in modo che il comportamento del robot meglio si adatti al caso studio.

2.1.2 Creazione di ostacoli dinamici

Per la generazione di ostacoli mobili si è dapprima fatto ricorso all'utilizzo di uno strumento noto in Gazebo come "actor" che grazie ad una proprietà di visualizzazione riusciva a simulare un ostacolo mobile, quale l'essere umano, in pose comuni e quotidiane come per esempio la camminata o la corsa, animando i movimenti. Il principale problema di questo tipo di oggetti risiede nella mancanza di caratteristiche di collisione, almeno nelle attuali versioni di Gazebo, che rendeva il lavoro di rilevamento degli stessi impossibile per il robot. Per questo motivo abbiamo sviluppato una sorta di "manichino" posto su un robot differenziale che si muove in maniera casuale all'interno dell'ambiente, seguendo percorsi in linea retta finché non incontra un ostacolo. In particolare i manichini sono stati dotati di un LIDAR 2D che rileva la distanza da eventuali ostacoli. Quando il soggetto si trova ad una distanza inferiore di 90cm, esso girerà di 30 verso destra e continuerà a muoversi. Il laser è stato posizionato in modo tale da rilevare anche la presenza del Pioneer, in quanto si è voluto simulare il caso in cui i soggetti non abbiano alcun interesse a collidere tra loro e quindi tutti gli attori in gioco si influenzano vicendevolmente.

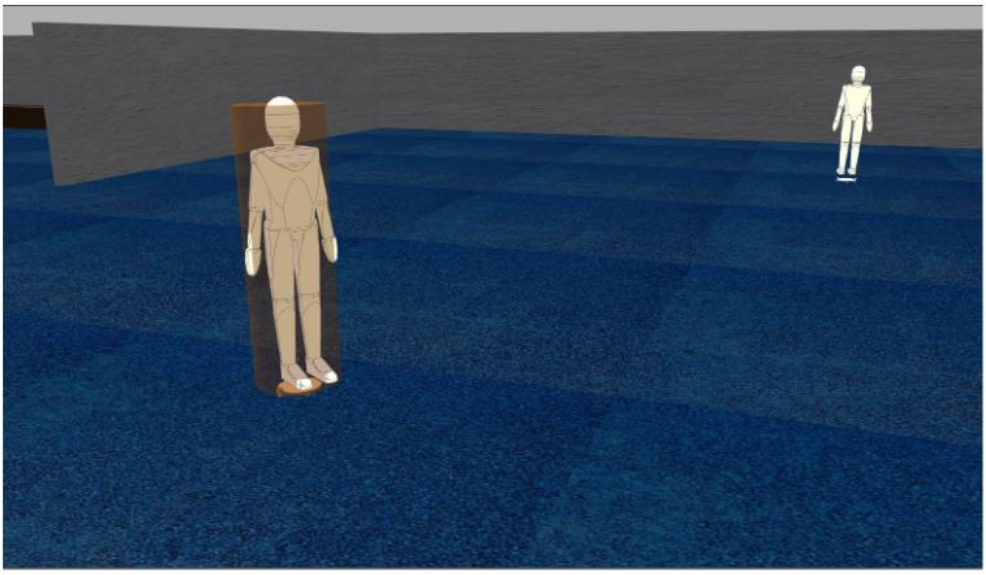


Figure 2.3: Rappresentazione del modello e della collision dei manichini

2.1.3 Ambiente di simulazione

Lo sviluppo del progetto ha richiesto il passaggio per diversi ambienti di simulazione prima di arrivare a quello definitivo in cui sono stati effettuati tutti i test.
Il primo ambiente era 20x10m e prevedeva unicamente la presenza di due manichini.

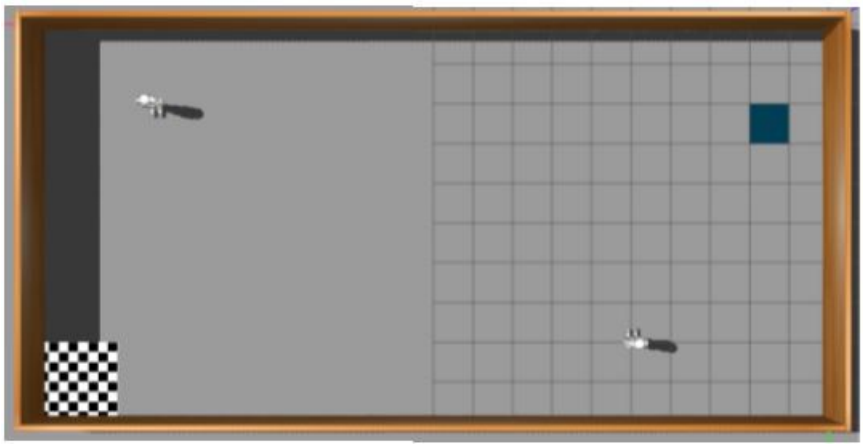


Figure 2.4: Ambiente di simulazione 10 x 20

Una volta ottenuti risultati soddisfacenti in questo ambiente, abbiamo deciso di modificarlo aumentando la complessità. Cioè sono stati aggiunti diversi complementi d'arredo ed anche una stanza.

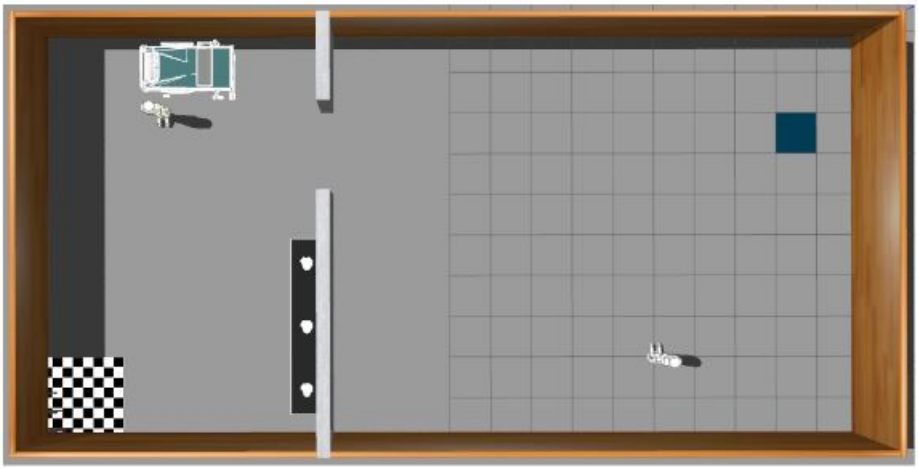


Figure 2.5: Ambiente di simulazione 10 x 20 con stanza

Successivamente, come già preannunciato in fase di introduzione, si è passare a dimensioni ben maggiori, 40 x 40 m. Anche in questo caso sono stati usati letti e scrivanie per rispecchiare quanto più possibile la realtà. Inoltre per poter coprire una così ampia superficie, il numero dei manichini è passato da due a sette. Per incrementare la difficoltà, come si può ben vedere, si è deciso di aumentare il numero di stanze a cui il robot deve poter accedere. La scelta delle dimensioni delle stanze e della loro posizione non è casuale ma è frutto della nostra volontà di testare il robot in un ambiente volutamente complesso e che rispecchiasse quanto più possibile uno scenario reale.

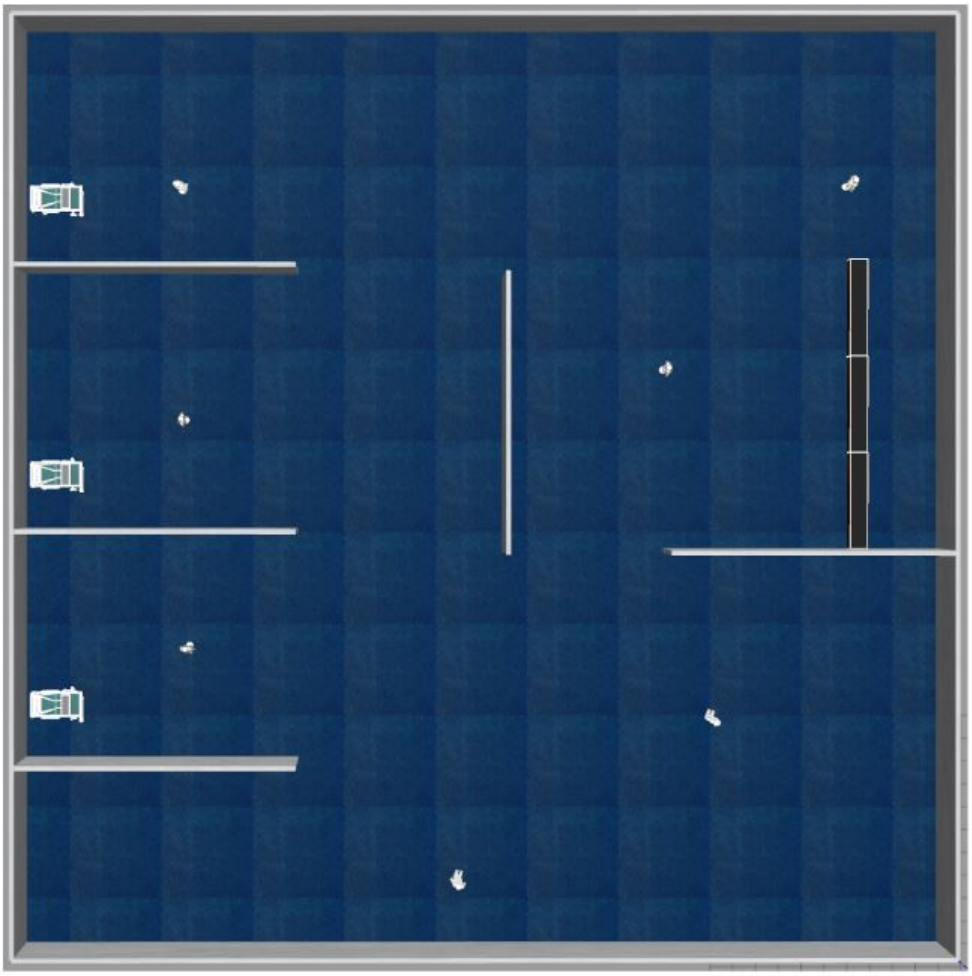


Figure 2.6: Ambiente di simulazione 40 x 40

2.2 Implementazione

Nel seguito saranno descritti tutti gli strumenti utilizzati e le soluzioni implementate.

2.2.1 OMPL

OMPL, o Open Motion Planning Library, raccoglie un vasto repertorio di algoritmi avanzati di pianificazione del moto basati su campionamento tra cui i tre pianificatori testati in questo progetto. A differenza degli altri strumenti impiegati in questo progetto, OMPL non è in alcun modo un package né ne esiste un wrapper per ROS(come invece esiste per il software di visione OpenCV) ma è una libreria stand-alone interamente scritta in C++.

2.2.2 Move_Base

È un pacchetto di ROS molto utilizzato nella navigazione di robot mobili. Fornisce la possibilità di spostare un robot, data una posizione iniziale, assegnandogli una posa desiderata con l'aiuto di altri nodi di navigazione. Tale nodo collega pianificatore globale e locale per calcolare un percorso valido. Nel caso in cui il robot sia bloccato o non trovi una soluzione, una routine di recovery compie un giro intorno all'asse verticale alla ricerca di una traiettoria percorribile. Le principali informazioni relative alla mappa, quali dimensioni e posizione degli ostacoli, sono generate dal nodo costmap2D gestito dal map-server. Questo processo avviene sia per la mappa globale sia per quella locale.
Per fornire a move_base una mappa dell'ambiente di lavoro si è utilizzato il package gmapping, fornito da ROS, per performare un algoritmo di SLAM. Durante la navigazione, la localizzazione del robot è stata affidata ad un altro package, AMCL (Adaptive Monte Carlo Localization), che sulla base dei dati odometrici, delle misurazioni del laser e della mappa precedentemente costruita, riesce a fornire una stima della posizione del robot nell'ambiente. Obiettivo dell'elaborato è stata la realizzazione di un plug-in per il pacchetto move_base che coniugasse l'ampia gamma di pianificatori offerti da OMPL con l'interfaccia e l'infrastruttura offerta da move_base. In tal modo è stato possibile sviluppare un plug-in versatile che con poche e semplici modifiche potesse eseguire svariati pianificatori.

2.2.3 Global_Planner Plugin

In questo paragrafo verranno analizzati i principali aspetti del plug-in sviluppato. La prima funzione che viene lanciata quando move_base carica il global_planner è quella di "initialize" in cui vengono istanziati tutti gli oggetti necessari.

```
void OmplGlobalPlanner::initialize(std::string name, costmap_2d::Costmap2DROS* costmap_ros)
{
    if (!_initialized)
    {
        ros::NodeHandle private_nh("~/" + name);
        _costmap_ros = costmap_ros;
        _frame_id = "map";
        _costmap_model = new base_local_planner::CostmapModel(*_costmap_ros->getCostmap());

        _plan_pub = private_nh.advertise<nav_msgs::Path>("plan", 1);
        private_nh.param("allow_unknown", _allow_unknown, true);

        global_costmap = *_costmap_ros->getCostmap();

        //get the tf prefix
        ros::NodeHandle prefix_nh;
        tf_prefix_ = tf::getPrefixParam(prefix_nh);

        _initialized = true;
        ROS_INFO("Ompl global planner initialized!");
    }
    else
    {
        ROS_WARN("This planner has already been initialized, you can't call it twice, doing nothing");
    }

}
```

Figure 2.7: Funzione Initialize

Quando l'utente fornisce un goal da raggiungere, viene chiamata la funzione "make_plane" la quale ha il compito di creare il path da eseguire per raggiungere l'obiettivo. In questa funzione vengono inizialmente impostati i vincoli all'interno dei quali dovrà essere ricercata una soluzione.

```
ob::RealVectorBounds bounds(2);
    bounds.setLow   (0, 0.0);
    bounds.setHigh  (0, map_x);
    bounds.setLow   (1, 0.0);
    bounds.setHigh  (1, map_y);
_se2_space->as<ob::SE2StateSpace>()->setBounds(bounds);

ob::RealVectorBounds velocity_bounds(1);
    velocity_bounds.setHigh ( 0.3);
    velocity_bounds.setLow  (-0.1);
_velocity_space->as<ob::RealVectorStateSpace>()->setBounds(velocity_bounds);

oc::ControlSpacePtr cspace(new oc::RealVectorControlSpace(_space, 2));
ob::RealVectorBounds cbounds(2);
    cbounds.setLow  (0, -0.04);
    cbounds.setHigh (0,  0.03);
    cbounds.setLow  (1, -0.02);
    cbounds.setHigh (1,  0.02);
cspace->as<oc::RealVectorControlSpace>()->setBounds(cbounds);
```

Figure 2.8: Definizione dei vincoli

Dopo viene creato l'oggetto "SpaceInformation" e ad esso viene associata la funzione "isStateValid", analizzata successivamente, che ha l'obiettivo di filtrare gli stati corretti da quelli che non lo sono.

```
oc::SpaceInformationPtr si(new oc::SpaceInformation(_space, cspace));
si->setStateValidityChecker(boost::bind(&OmplGlobalPlanner::isStateValid, this, si.get(), _1));
```

Figure 2.9: Definizione di SpaceInformation

A questo punto si instanzia l'oggetto "ProblemDefinition", il quale si occupa di tenere in considerazione non solo i vincoli dello spazio di stato, ma anche eventuali funzioni di ottimizzazione. Inoltre, per poter produrre una soluzione è necessario conoscere lo start state e il goal state ed anche il range del goal con il quale poter considerare una soluzione un percorso valido. Tale valore è stato identificato come "Goal Range" e risulta il terzo parametro in input alla funzione "setStartAndGoalStates". Tutti i parametri verranno meglio spiegati ed analizzati nel capitolo successivo.

```
ob::ProblemDefinitionPtr pdef(new ob::ProblemDefinition(si));
pdef->setStartAndGoalStates(ompl_start, ompl_goal, 5.0);
pdef->setOptimizationObjective(0.0001*length_objective);
```

Figure 2.10: Definizione di ProblemDefinition

Il passo successivo è quello di definire l'algoritmo utilizzato per la ricerca della soluzione e la conseguente impostazione dei suoi parametri.

```
//og::RRT* pointer = new og::RRT(si);
//og::pRRT* pointer = new og::pRRT(si);
og::RRTstar* pointer = new og::RRTstar(si);

//Planner Setting
pointer->setRange(dist_to_goal+10);
pointer->setGoalBias(0.5);

ob::PlannerPtr planner(pointer);
planner->setProblemDefinition(pdef);
planner->setup();
```

Figure 2.11: Definizione dell'algoritmo di soluzione

Una volta che tutti i parametri sono stati definiti, è possibile chiamare la funzione "solve" del planner per ricercare una soluzione. Nel caso in esame, il tempo impostato per la prima soluzione è di 20 secondi, in modo da permette la ricerca di una soluzione valida. Per il resto dell'esecuzione il tempo viene ridotto ad un secondo, in quanto si vuole che il robot possa essere quanto più reattivo possibile ad eventuali ostacoli.

```
if (first_plan){
  solved = planner->solve(20.0);
  first_plan = false;
} else {
  solved = planner->solve(1.0);
}
```

Figure 2.12: Chiamata alla funzione solve()

L'ultima azione consiste nel convertire la soluzione trovata dall'algoritmo, in un vettore di tipo "PoseStamped" fruibile da move_base.

```
if (solved)
{
    ROS_INFO("Ompl done!");
    ob::PathPtr result_path1 = pdef->getSolutionPath();

    // Cast path into geometric path:
    og::PathGeometric& result_path = static_cast<og::PathGeometric&>(*result_path1);

    // Create path:
    plan.push_back(start);

    // Conversion loop from states to messages:
    std::vector<ob::State*>& result_states = result_path.getStates();
    for (std::vector<ob::State*>::iterator it = result_states.begin(); it != result_states.end(); ++it)
    {
        // Get the data from the state:
        double x, y, theta, velocity;
        get_xy_theta_v(*it, x, y, theta, velocity);

        // Place data into the pose:
        geometry_msgs::PoseStamped ps = goal;
        ps.header.stamp = ros::Time::now();
        ps.pose.position.x = x;
        ps.pose.position.y = y;
        plan.push_back(ps);
    }

    plan.push_back(goal);
}

return !plan.empty();
```

Figure 2.13: Conversione della soluzione per essere utilizzata da move_base

A seguire la porzione di codice relativa alla funzione “isStateValid”.I controlli effettuati riguardano il rispetto dei vincoli impostati in fase di definizione del problema (“Check bound”) , la posizione degli stati interni alla mappa (“Check map dimension”), la posizione di stati che non siano interni o vicini ad ostacoli (“Check point validity”) e che due stati nella mappa locale non siano collegati con un tratto che vada ad intersecare un ostacolo (“Check path validity”).

```
bool OmplGlobalPlanner::isStateValid(const oc::SpaceInformation *si, const ob::State *state)
{
    double x, y, theta, v = 0.0;
    double cost = 0.0;
    double resolution = global_costmap.getResolution();

    //Check bound
    if (!si->satisfiesBounds(state)) {
        return false;
    }

    //Check map dimension
    get_xy_theta_v(state, x, y, theta, v);
    if ((x>map_x) || (y>map_y) || (x<0) || (y<0) ) { return false; }

    //Point Validity
    cost = _costmap_model->pointCost(x/resolution, y/resolution);
    if (cost < 0 ) { return false; }

    //Path Validity
    _costmap_ros->getRobotPose(curr_pose);
    cost = _costmap_model->lineCost(curr_pose.pose.position.x/resolution, x/resolution,
                                    curr_pose.pose.position.y/resolution, y/resolution);

    if (cost < 0) { return false; }
}
```

Figure 2.14: Funzione isStateValid

2.2.4 Dynamic Obstacles Avoidance

L'implementazione del comportamento di Dynamic Obstacles Avoidance nasce dalla possibilità di unire le informazioni provenienti da diversi fonti: la local costmap, gli stati prodotti dall'algoritmo risolutivo e la posizione del robot. La logica che abbiamo usa i dati degli ostacoli provenienti dalla dalla mappa locale e vengono considerati inaccessibili tutti gli stati che collegati in linea retta con l'attuale posizione del robot intersecano un ostacolo. In tal modo, il robot quando incontra un ostacolo è portato a riconsiderare lo stato che stava per raggiungere in favore di uno che rispetti questo vincolo e quindi che possa raggiungere in totale sicurezza.

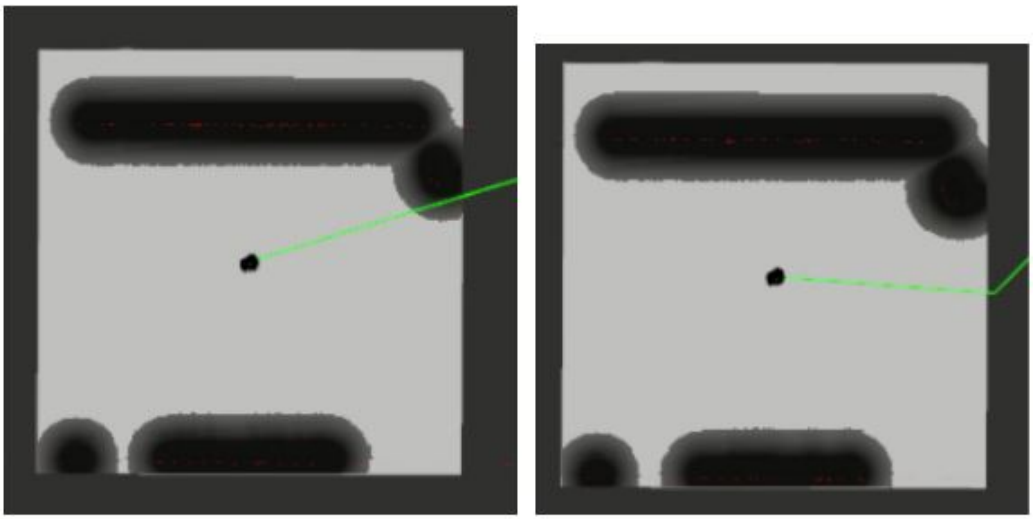


Figure 2.15: Esempio di Dynamic Obstacles Avoidance

Come si può notare dalle immagini, appena il robot identifica che all'interno della sua local costmap il path interseca un ostacolo, al ciclo successivo genererà un soluzione per evitarlo.

Chapter 3

Test e Risultati

Una volta sviluppato il plugin e determinato l'ambiente della simulazione abbiamo realizzato diversi test in vari scenari e con diversi valori caratteristici dei tre pianificatori implementati, analizzando i risultati e confrontandoli per determinare l'algoritmo più adatto per ogni situazione.
Nella serie di svariati test, sei di questi rappresentano gli esempi più esplicativi dei risultati finora raggiunti. Particolare attenzione è stata rivolta a cinque fattori e alla loro manipolazione:

- **Goal Range:** rappresenta la threshold associata al goal per la quale se uno stato che cade all'interno di questa zona, può dare luogo ad una soluzione

- **Solve Time**: tempo di pianificazione.

- **Funzione di ottimizzazione** : le due scelte fatte non sono altro che somma di due fattori definiti "Cost_objective" e "Length_objective" pesati opportunamente o la scelta di uno solo dei due.

- **Range**: rappresenta il massimo valore del passo tra un nodo e il successivo nell'albero dei nodi.

- **Goal Bias** : la variazione tra 0 e 1 di tale parametro ha determinato una pianificazione orientata maggiormente al goal generando traiettorie sempre più vicine ad un segmento di linea retta tra punto di partenza e obiettivo.

3.0.1 Scelta del test

Ogni singola simulazione è stata replicata alla stessa maniera imponendo al robot il raggiungimento di due obiettivi principali in un due stanze diverse della mappa definiti dai punti P1 e P2. Ogni caso studio è stato ripetuto per dieci volte e di seguito sono riportati i dati maggiormente rilevanti per ognuno degli algoritmi. Di seguito saranno confrontati i tre algoritmi e i loro risultati in tre configurazioni parametriche differenti, di cui una definita "standard" in quanto fornita dalla libreria OMPL. I goal da raggiungere (P1 e P2) sono stati scelti in modo da mettere alla prova la capacità dell'algoritmo di generare due tipologie di traiettorie strutturalmente diverse. Per P1, come risulta evidente dalla mappa (fig. 3.1), sebbene la lunghezza della traiettoria sia relativamente bassa, il robot si troverà ad affrontare un angolo di curvatura particolarmente pronunciato unito alla possibilità di incappare fin da subito nel problema del dynamic obstacle avoidance comportando quindi un allontanamento dalla traiettoria desiderata che dovrà essere corretto in seguito con comportamenti maggiormente orientati al raggiungimento del goal. Diversamente il punto P2 dà la possibilità al robot di affrontare con maggiore probabilità un ostacolo dinamico contemporaneamente al fatto che dovrà correttamente valutare un percorso che attraversi il varco per la camera centrale. Tale scelta è stata effettuata sulla base di risultati altamente inattesi nelle prime fasi di progetto in cui il robot definiva soluzioni accettabili anche quelle che intersecavano le pareti, chiaramente non ammissibili.

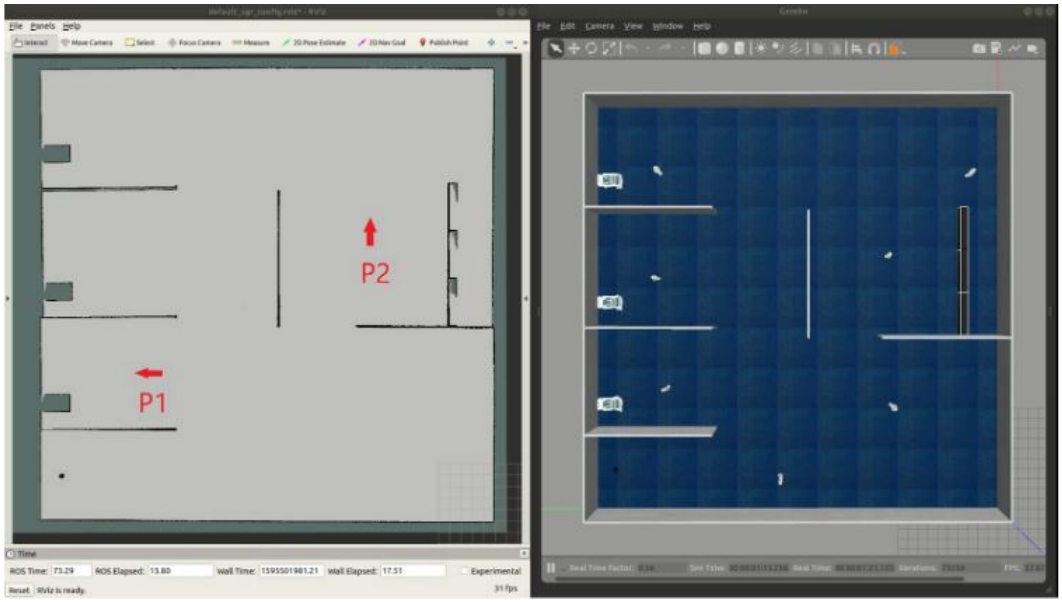


Figure 3.1: Posizione dei punti P1 e P2 usati nei test

	Optimization	Range	Goal Range	Solve Time
Standard Config.	10e-4 Length_Objective	dist_to_goal + 10	5.0	20 / 1
Algoritmi (P1 / P2)	Real Solve Time	Path Quality	Notes	
RRT*	20s / 20s	Buona / Buona		
pRTT	1s / 20s	Pessima / Pessima		
RRT	3s / 20s	Pessima / Pessima		

Test 1

	Optimization	Range	Goal Range	Solve Time
Test 1	Cost_Objective + Length_Objective	dist_to_goal + 10	5.0	20 / 1

Algoritmi (P1 / P2)	Real Solve Time	Path Quality	Notes
RRT*	20s / 20s	Buona / Buona	Per P1 e P2 le attraietorie approssimate intersecano gli ostacoli
pRTT	1s / 20s	Pessima / Pessima	Nessuna incertezza nel tratto finale
RRT	1s / 20s	Pessima / Pessima	Nessuna incertezza nel tratto finale

Test 2

	Optimization	Range	Goal Range	Solve Time
Test 2	10e-4 Length_Objective	5	5.0	20 / 1

Algoritmi (P1 / P2)	Real Solve Time	Path Quality	Notes
RRT*	20s / 20s	Buona / Ottima	Path migliori della configurazione standard, ma cresce l'incertezza nel tratto finale
pRTT	20s / 20s	Buona / Discreta	Path migliori della configurazione standard, ma cresce leggermente l'incertezza nel tratto finale. Nel P2 i path hanno forma più irregolare rispetto P1 data la maggiore distanza.
RRT	3s / 20s	Discrteta / Discreta	La qualità dei path migliora leggermente, ma cresce anche (di poco) l'incertezza nel tratto finale.

Test 3

	Optimization	Range	Goal Range	Solve Time
Test 3	10e-4 Length_Objective	50	5.0	20 / 1

Algoritmi (P1 / P2)	Real Solve Time	Path Quality	Notes
RRT*	20s / 20s	Buona / Buona	Incertezza nel tratto finale di P2
pRTT	2s / 20s	Pessima / Pessima	

Test 4

	Optimization	Range	Goal Range	Solve Time
Test 4	10e-4 Length_Objective	dist_to_goal + 10	5.0	1

Algoritmi (P1 / P2)	Real Solve Time	Path Quality	Notes
RRT*	X	Buona / Buona	Incertezza nel tratto finale di P2 e P1
pRTT	X	Pessima / Pessima	
RRT	X	Pessima / Pessima	

Test 5

	Optimization	Range	Goal Range	Solve Time
Test 5	10e-4 Length_Objective	dist_to_goal + 10	0.1	20 / 1

Algoritmi (P1 / P2)	Real Solve Time	Path Quality	Notes
RRT*	20s / 20s	Buona / Buona	Per P1 le traiettorie intersecano gli ostacoli
pRTT	20s / 20s	Pessima / Pessima	

Test 6

	Optimization	Range	Goal Range	Solve Time
Test 6	10e-4 Length_Objective	dist_to_goal + 10	10	20 / 1

Algoritmi (P1 / P2)	Real Solve Time	Path Quality	Notes
RRT*	20s / 20s	Pessima / Buona	Le treiettorie intersecano gli ostacoli per P1 e il robot non riesce a raggiungere il goal. Per P2 il robot ha difficoltà a raggiungere il goal nel tratto finale
pRTT	1s / 1s	Pessima / Pessima	Le traiettorie intersecano gli ostacoli per P1 e P2.
RRT	1s / 1s	Pessima / Pessima	Le traiettorie intersecano gli ostacoli per P1 e P2. Difficoltà a trovare una soluzione valida.

Conclusioni

Dai test effettuati risulta come era ragionevole intuire a priori che i risultati migliori in questo ambiente si ottengono sfruttando l'algoritmo RRT* ma non banale è stata la derivazione di quale sia la scelta migliore dei parametri. In primo luogo uno dei parametri che è stato fissato ad un valore costante in tutte le simulazioni è stato il GoalBias, i risultati ottenuti da un aumento di tale parametro risultano caratterizzati da un andamento pressoché costante sviluppato quasi in linea retta tra punto di partenza e arrivo, tale proprietà riduce i tempi di elaborazione di una soluzione ma porta in secondo piano un fattore particolarmente rilevante quale obstacle-avoidance di oggetti statici e dinamici e, in certe circostanze anche la realizzazione di questo comportamento mantenendo quella che si potrebbe definire una "distanza di sicurezza". Pertanto si è cercato un valore che realizzasse un giusto trade-off tra orientamento al goal e pianificazione di path sicuri. Due parametri sono stati caratterizzati in funzione delle specifiche richieste nella formulazione di ogni "richiesta" formulata al move_base server. In particolare, dato l'ambiente dinamico e la proprietà di reiterazione della pianificazione di percorso attuata dal nodo move_base si è scelto di dare al parametro Solve Time una forte flessibilità che risulta in una prima pianificazione relativamente grande e in qualche modo proporzionale alla distanza dell'obiettivo e un tempo sicuramente più breve in tutte le iterazioni successive che permettesse sia di preservare la dinamicità e prontezza del robot, sia di "aggiustare il tiro" durante la simulazione. Secondo parametro a cui ci si riferiva è il Range a cui è stata dapprima conferita una diretta proporzionalità alla distanza in linea d'aria tra partenza e traguardo, con talvolta un offset a maggiorare tale valore. I risultati ottenuti però con RRT e pRRT non garantivano la stessa qualità di un valore fisso sufficientemente basso che resta quindi la scelta migliore per questi due algoritmi. Infine con un processo puramente di trial-and-error si è proceduto a determinare i giusti valori di Goal Range e la giusta funzione obiettivo, peculiarità principale di ogni algoritmo di pianificazione.

Bibliografia

[1] Paolo Ferrari, Marco Cognetti, and Giuseppe Oriolo. Anytime wholebody planning/replanning for humanoid robots. 2018 IEEE-RAS 18th International Conference on Humanoid Robots (Humanoids), 2018.

[2] Sertac Karaman and Emilio Frazzoli. Incremental sampling-based algorithms for optimal motion planning. Robotics Science and Systems VI, 104:2, 2010.

[3] Sertac Karaman and Emilio Frazzoli. Sampling-based algorithms for optimal motion planning. The international journal of robotics research, 30(7):846–894, 2011.

[4] Sertac Karaman, Matthew R Walter, Alejandro Perez, Emilio Frazzoli, and Seth Teller. Anytime motion planning using the rrt. Institute of Electrical and Electronics Engineers, 2011.

[5] Steven M LaValle and James J Kuffner Jr. Randomized kinodynamic planning. The international journal of robotics research, 20(5):378–400, 2001.

[6] John Reif. Complexity of the mover's problem and generalizations. Proc. IEEE Symp. on Foundations of Computer Science, 1979.

[7] P Svestka, JC Latombe, and LE Overmars Kavraki. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. IEEE Transactions on Robotics and Automation, 12(4):566–580, 1996.